DAGR: A DSL for Legacy OpenCL Codes

Keir, Paul

Published: 13/03/2016

Document Version
Publisher's PDF, also known as Version of record

DAGR: A DSL for Legacy OpenCL Codes

Porting SLAMBench KFusion to SYCL

Paul Keir
University of the West of Scotland
paul.keir@uws.ac.uk

Abstract

The C++ SYCL for OpenCL standard was ratified in 2015 by the Khronos Group, with early commercial and open-source implementations available already. Traditional OpenCL developers curious about the possibilities of the new API will find there is much to discover; including new C++ `accessor` classes, and the use of lambda functions and function objects. In this paper we introduce the DAGR embedded domain specific language, which attempts to provide an interface which can be readily used and understood by established OpenCL users; and benefit those porting OpenCL codes to SYCL especially. To investigate the robustness of SYCL and its implementations, as well as to help design and evaluate the DAGR API, we report on the completed effort to port the SLAMBench KFusion computer vision benchmark to SYCL using DAGR.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming; D.1.5 [*Programming Techniques*]: Object-oriented Programming

Keywords GPGPU, OpenCL, C++, Parallelism, Computer Vision

1. Introduction

The Khronos SYCL 1.2 specification [4] defines a C++11/14 API for parallel programs targeting execution upon a wide range of heterogeneous and hierarchical processing hardware of the kind typically targeted by OpenCL. Distinct from the pending OpenCL C++ kernel language, the foremost characteristic of SYCL is the complete integration of host and device code; wherein a kernel specified by a function object can as readily be passed to a SYCL `parallel_for` function template, as called directly. So too, an arbitrary function may as well be called within the call-graph of a kernel targeting the OpenCL device; as within common or garden host code. Another interesting aspect of SYCL concerns the transparent exchange of C++ type information between host and device; permitting novel possibilities for GPGPU template metaprograms. With no extensions to the C++ language, it is also reassuring to find that a serial execution is always at hand for debugging; to borrow the language of OpenMP, a SYCL program is *single-source*.

Yet a SYCL program has its "boilerplate". Understandably, a pointer to host memory cannot be used within device code. Consequently, the SYCL runtime must be provided with the category and extent of a memory region; along with notification of access permissions, such as `read_write`. The recommended practice is the creation of a `buffer` object for each such address or pointer. Before launching a kernel, a command group function object must then be created, and scheduled by its provision as an argument to the `commit` method of a SYCL `queue`. The call operator of this function object must then create an `accessor` object for each `buffer` object; before the final kernel launch command is issued, by invoking a method of the `handler` argument, such as `parallel_for`. Besides verbosity, there is of course scope for user error; for example, variable declarations in the wrong scope, or incorrect lambda *capture-defaults*. The DAGR header library addresses such concerns.

```
__kernel void vec_add(__global const float *a,
                      __global const float *b,
                      __global       float *c,
                      const size_t sz) {
  size_t id = get_global_id(0);

  if (id < sz)
    c[id] = a[id] + b[id];
}
```

Figure 1. An OpenCL C vector addition kernel

Consider the OpenCL C vector addition kernel in Figure 1 as a reference point, in anticipation of an equivalent within the DAGR API. SYCL is an ideal solution for introducing OpenCL-style parallelism to a serial or homogeneously parallel C++ project; yet of course there now exist a sizeable quantity of GPGPU codes using CUDA or OpenCL. When porting legacy OpenCL codes to SYCL, the calls to OpenCL C built-in functions can be transposed directly to those of SYCL; with many types sharing the same names.

```
struct vec_add {
  template <typename I, typename T>
  static void k(I ix,
                const T *a, const T *b, T *c,
                const size_t extent) {
    size_t id = ix[0];

    if (id < extent)
      c[id] = a[id] + b[id];
  }
};
```

Figure 2. A SYCL vector addition kernel for DAGR

SYCL 2016, March 12–16, 2016, Barcelona, Catalonia, Spain.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

The primary goal of the DAGR header library is better support for the *cut and paste* transplant of kernels written in OpenCL C, into SYCL C++. Consequently, DAGR supports comparable function signatures; Figure 2 demonstrates the DAGR equivalent of Figure 1. Conventionally, SYCL `accessor` objects provide the crucial working abstraction over host memory; yet while some pointer interface operators such as `operator*` and `operator[]` *are* supported, others, such as `operator++` and `operator--` are not. A SYCL `accessor` satisfies the *dereferenceable* concept, but not the *iterator* concept. This is true also of SYCL's explicit pointer classes; such as `global_ptr`. It is no surprise then to find that a SYCL `accessor` object cannot be passed as an argument to a polymorphic pointer parameter such as `T*`; and of course the `std::is_pointer` trait also evaluates to `false`.

```
template <typename T>
void vec_add_h(const T *a, const T *b, T *c,
               const size_t sz)
{
  queue q;
  const range<1> r(sz);
  const buffer<T,1> buf_a(a, r);
  const buffer<T,1> buf_b(b, r);
        buffer<T,1> buf_c(c, r);

  dagr::run<vec_add>(q,r,buf_a,buf_b,buf_c,sz);
}
```

Figure 3. SYCL vector addition host code for DAGR

Another common aspect of OpenCL kernel function signatures which DAGR mirrors in SYCL is in the correspondence of kernel value parameters with those host variables which do not originate from SYCL `buffer` objects. This is illustrated by the `sz` argument to the `dagr::run` method in Figure 3, which corresponds to the `size_t` value parameter of the kernel in Figure 2.

The DAGR API is subsequently applied to port version 1.1 of the main KFusion algorithm [12] of the SLAMBench computer vision benchmark suite [1, 2] to SYCL. SLAMBench provides implementations of KFusion in a range of languages, including an OpenCL version, which naturally provides the starting point here. Nevertheless, the final result of the port is more akin to the C++ version, upon which the OpenCL version was originally based. Development was undertaken using a trial version of Codeplay Software's *ComputeCpp* [8]. Accordingly, performance evaluation of the SYCL port of SLAMBench is deferred.

2. DAGR Before Thee

Let us turn to consider some design choices of the DAGR API; implemented as a single 400-line header file. Figures 2 & 3 already invite comment. The kernel itself is represented as a C++ class with a `static` member method, which should currently have a fixed name: `k`; for brevity. This is a departure from the idiomatic use of C++ lambda expressions. Yet when reckoning with the port of an OpenCL project, it is clear that transplanting each, possibly large, OpenCL kernel to the site of its enqueueing may be unwieldy, and will likely need refactoring. The choice then of a `static` method simply removes the requirement for either the user or implementation to instantiate the class at runtime. A function object can of course offer more flexibility, allowing the context, the class members, to be modified from one kernel launch to the next. Lacking this feature does not though compromise DAGR's goal to facilitate an OpenCL project transition, and its context-free OpenCL C kernels. There is also the option to use a function template to define the kernel; though this approach will require the user to explicitly specify the type of each kernel argument.

```
template <class K, class R, class ...Xs>
void run(cl::sycl::queue, const R, Xs &&...);
```

Figure 4. Signature of DAGR's main function

The signature for the main DAGR variadic function template, `dagr::run`, to enqueue and invoke a kernel, is shown in Figure 4. The first template parameter of `dagr::run` identifies a class which must contain a `static` method `k`, defining the kernel; and is the only template parameter which must be provided explicitly.

Regarding runtime parameters, `dagr::run` expects a SYCL `queue` as its first argument; and a SYCL `range` or `nd_range`, used by the internal call to `parallel_for`, as its second. The function parameter pack relating to `Xs` supplies the arguments for the `K::k` kernel method; where parameters specified as pointers, should originate from SYCL `buffer` object arguments, or derivatives. This is demonstrated in Figure 3, where the three `buffer` arguments, `buf_a`, `buf_b` and `buf_c`, provided to `dagr::run`, correspond to the three pointer parameters of Figure 2's `vec_add::k` method. The `sz` argument to `dagr::run` in turn corresponds to the `size_t` value parameter.

The first parameter of each DAGR kernel specified by a `static` class method `k` is reserved for the kernel index; corresponding to an `item` or `nd_item`, for a `range` or `nd_range` argument to `dagr::run` respectively.

2.1 Tagging Buffers

SYCL allows the same buffer to be used in multiple kernels, each with differing write access permissions. This is configured by the parameters used in the creation of each SYCL `accessor` within a command group functor. DAGR currently supports a subset of this functionality through the application of tag classes, which exist solely to associate compile-time information with their sole member; in this case, a SYCL `buffer`.

```
template <class T> inline ro_tag_t<T> ro(T &&t);
template <class T> inline wo_tag_t<T> wo(T &&t);
```

Figure 5. Read and write permission wrapper functions

Figure 5 lists the signature of two DAGR wrapper functions. Given a buffer argument, the result of either can also be used as an argument to the `dagr::run` function, each configuring the creation of an `accessor` object, with `access::mode::read` or `access::mode::write` access permission. An untagged `buffer` will request `access::mode::read_write` access, unless it is `const`, in which case it too will request `access::mode::read`.

```
struct zero {
  template <typename I, typename T>
  static void k(const I ix,              T *y) {
    y[ix[0]] = 0;
  }
};

struct copy {
  template <typename I, typename T>
  static void k(const I ix, T *x, const T *y) {
    x[ix[0]] = y[ix[0]];
  }
};
```

Figure 6. Two separate DAGR kernels

To illustrate their use, consider the two DAGR kernels shown in Figure 6. In a somewhat contrived fashion, via two separate kernel launches, we seek to first "zero" a `buffer` accessed through the `y` parameter of `zero::k`; before assigning each value of a second `buffer`, accessed through the `x` parameter of `copy::k`, to that of the first `buffer`. Figure 7 demonstrates the host code to facilitate such an affair.

Observe firstly Figure 7's declaration of a `buffer` with storage managed by the SYCL runtime: `buf_d`. The SYCL runtime can likely store this entirely on the device. The first call to `dagr::run` then sees the `wo` wrapper function deployed to indicate that this `buffer` will only be written to. The second call to `dagr::run` applies the `ro` wrapper function to the same `buffer`, to indicate that `access::mode::read` access is now sufficient. Meanwhile the `buf` argument, which was created using a host address, is wrapped by a call to `wo`. Once `buf` is destroyed, the result is copied back to `a`.

Of course a `buffer` can also be represented using a C++ temporary value: replacing `wo(buf)` with `wo(buffer<T,1>(a, range<1>{sz}))` in Figure 7 even returns the data to host memory at `a` marginally sooner; as the call to `dagr::run` completes.

```
template <typename T>
void copy_h(T *a, const size_t sz)
{
  queue q;
  const range<1> r(sz);
  buffer<T,1> buf_d(  r);
  buffer<T,1> buf  (a,r);

  dagr::run<zero>(q,r,         wo(buf_d));
  dagr::run<copy>(q,r,wo(buf),ro(buf_d));
}
```

Figure 7. Two-stage zeroing via a device-side `buffer`

2.2 Local Memory Tags

Programming with OpenCL local memory in SYCL begins with the creation of an `accessor` object, and a target parameter of `cl::sycl::access::target::local`. Within DAGR we seek means to specify the quantity of local memory in a typeful manner, allowing an appropriate encoding object also to function as an argument to `dagr::run`. For brevity, only the one-dimensional function template for `lo` is shown in Figure 8. A call to `lo` requires both the compile-time element type `T`; and the runtime quantity `x`. Binary and ternary overloads similarly produce `local_t<T,2>` and `local_t<T,3>` values respectively.

```
template <typename T>
inline local_t<T,1> lo(const size_t x);
```

Figure 8. Lightweight local memory specification

Figure 9 provides a DAGR example using local memory so. The `rev_local` kernel is a straightforward permutation exercise involving the `barrier` method of `nd_item` on line 9, equivalent to OpenCL's workgroup barrier; though note that `U` and `T` will have different types, due to their distinct address spaces. The host code function `rev_local_h`, meanwhile, demonstrates the use of `lo` to request a contiguous region of `sz/2` elements of type `T` in local memory.

```
1  struct rev_local {
2    template <class I, class T, class U>
3    static void k(I ix, T *p, U *loc) {
4      const auto   lr = ix.get_local_range(0);
5      const auto  lid = ix.get_local()[0];
6      const auto  gid = ix.get_global()[0];
7
8      loc[lid] = gid;
9      ix.barrier(access::fence_space::local);
10     p[gid]   = loc[lr - lid - 1];
11   }
12 };
13
14 template <typename T>
15 void rev_local_h(T *a, const size_t sz)
16 {
17   queue q;
18   nd_range<1> ndr(range<1>{sz},range<1>{sz/8});
19   buffer<T,1> buf_a(a,range<1>{sz});
20
21   dagr::run<rev_local>(q,ndr,buf_a,lo<T>(sz/8));
22 }
```

Figure 9. Example using local memory in DAGR

3. SYCL SLAMBench

SLAMBench [1, 2] is a computer vision benchmark suite providing implementations of the KFusion algorithm [12], to solve the simultaneous localisation and mapping (SLAM) problem, given colour and depth information; i.e. RGB-D. KFusion, which in fact uses only the depth information, is an open-source implementation of the KinectFusion [11] algorithm, provided by SLAMBench in a range of C-based languages and APIs; including serial C++, CUDA, OpenCL and OpenMP. The ICL-NUIM dataset [10] of synthetic RGB-D sequences provides a reference input upon which each implementation's performance and accuracy can be examined.

The SYCL version of the SLAMBench KFusion algorithm was developed and tested on 64-bit Ubuntu 15.04, with GCC 4.9.2, and the 15.06 and 15.10 evaluation versions of Codeplay Software's *ComputeCpp*. The OpenCL driver employed was version 5.0.0.43 from the 64-bit Intel Code Builder for OpenCL. A significant first task was the development of a GCC-like compiler driver, `syclcc`, which automatically invokes both the device compiler from Codeplay's *ComputeCpp*, *and* the native C++ host compiler. So equipped, and following minor modifications to the SLAMBench CMake configuration, by setting `CXX` to `syclcc`, the GUI and benchmark versions of KFusion will build; upon the same simple `cmake` and `make` command invocations as for each supported language.

Version 1.1 of SLAMBench includes 14 significant kernels. Of these, 12 are implemented on GPU using both CUDA and OpenCL. The two remaining are *acquire*, the IO-heavy acquisition of another RGB-D frame; and *solve*, a singular value decomposition, too small to be offloaded, which utilises an external library: TooN [9]. So too, the SYCL version contains the same 12 kernels.

The focus for the original SLAMBench project is portability; with a nevertheless competitive performance profile. From this perspective a goal in the development of the SYCL version was in replicating the high-level structure of the KFusion algorithm implementation; which is shared by each of the language implementations included with SLAMBench. Due to the Khronos specified compatibility between OpenCL and SYCL, the OpenCL implementation was though the foundation in the development of the SYCL equivalent. This meant that OpenCL `cl_mem` variables declared at global scope, became SYCL `buffer` pointers; also declared at global scope. Calls to `clCreateBuffer` became calls to the C++ `new` operator, with some care required as the relevant read/write access permission is not requested by the SYCL `buffer` constructor; using a subsequent SYCL `accessor` instead. Hence,

access permission requests were transferred to the site of each kernel enqueue. The declaration; creation; and release of kernels are implicit in SYCL, and were thus elided in this version. Calls to `clSetKernelArg` and `clEnqueueNDRangeKernel` are rendered especially concise with a single call to `dagr::run`. Ultimately, the aim in porting the host component in this fashion is readability, so allowing a fruitful comparison of the SYCL and existing versions; and parity of performance by replicating the same algorithm.

Figure 10's invocation of the *bilateralFilter* kernel in SYCL, uses the DAGR API, with the `ro` wrapper function utilised twice to request `access::mode::read` access. The single call to `dagr::run` shown occurs in the `Kfusion::preprocessing` method, and is directly comparable to the single call to `bilateralFilterKernel` in the C++ version; albeit with additional SYCL `queue` and `range` arguments. In the more verbose OpenCL version, `clSetKernelArg` is called for each of five arguments, before the call to `clEnqueueNDRangeKernel`.

```
dagr::run<bilateralFilterKernel>(q, r,
  *ocl_ScaledDepth[0], ro(*ocl_FloatDepth),
  ro(*ocl_gaussian), e_delta, radius);
```

Figure 10. DAGR enqueue of the bilateral filter kernel

4. Related Work

Regarding purely C++ interfaces to GPU acceleration, the obvious reference is SYCL [4] itself. Meanwhile, a published ISO standard in consideration for inclusion in the next iteration of the C++ standard [5] provides an API for the parallel execution of a range of common STL algorithms by the addition of tag-like *execution policy* objects as the first argument to the relevant algorithm functions; with sequential, parallel and vectorised (including GPU) implementations specified. A SYCL implementation [7] also exists; with 8 STL algorithms implemented so far. Along similar lines, NVIDIA's proprietary Thrust library [13] provides an extensive STL-like library with a purely C++ interface to an underlying NVIDIA GPU-only CUDA layer.

Another example of a DSL using SYCL is described in Potter [3]. That research contrasts with the current work in its use of a *deep* embedding, to allow the composition of kernels. ViennaCL [14] provides a C++ API targeting a range of backends including OpenMP, OpenCL and CUDA. The main focus for ViennaCL is the provision of common linear algebra operations, with OpenCL strings facilitating custom kernels via runtime compilation.

5. Conclusion

The SYCL DAGR API has been introduced, along with its deployment to the task of porting the SLAMBench computer vision benchmark suite [1, 2] to SYCL, for compatibility with Codeplay Software's implementation: *ComputeCpp*. Notable features of DAGR include a concise end-user API; support for both value and pointer kernel parameters, as in OpenCL C; and the provision of lightweight wrapper functions to convey information regarding access permissions and shared memory quotas to the DAGR implementation. The API is designed as a header-only library, and utilises the Khronos SYCL standard to provide a concise interface for developers less interested in C++ lambda functions per se, than in maintaining parity between an OpenCL and a SYCL backend. While it is understood that the DAGR API offers only a section of SYCL's configuration space, it is hoped that this is nevertheless a focused and useful portion.

Bearing in mind that SYCL was only ratified in 2015, it is pertinent to mention that DAGR also represents a lightweight portability layer to accommodate the different stages of development between both the OpenCL-compatible *ComputeCpp* from Codeplay Software; and the open source *triSYCL* [6] implementation from Ronan Keryell.

Future work will aim to introduce *triSYCL* support; first for DAGR, and then for SYCL SLAMBench. Regarding further features in DAGR, the following are prioritised:

- *Write* access to `dagr::run` arguments, without an explicit `buffer` wrapper;
- Support work-item built-in functions; e.g. `get_global_id`;
- Allow the user to name the kernel method; i.e. not always `k`;
- Provide a SYCL `single_task` overload for `dagr::run`

Acknowledgments

Thanks to Ruyman Reyes, Maria Rovatsou and Uwe Dolinsky from Codeplay Software's *ComputeCpp* development team for their excellent support and quick response to queries. I am also grateful for the efforts of Ronan Keryell in establishing an open-source SYCL implementation so quickly.

References

[1] M. Z. Zia, L. Nardi, A. Jack, E. Vespa, B. Bodin, P. H. J. Kelly and A. J. Davisonr, *Comparative Design Space Exploration of Dense and Semi-Dense SLAM*, IEEE Intl. Conf. on Robotics and Automation (ICRA 2016), Stockholm, Sweden, May 2016.

[2] L. Nardi, B. Bodin, M. Z. Zia, J. Mawer, A. Nisbet, P. H. J. Kelly, A. J. Davison, M. Luján, M. F. P. O'Boyle, G. Riley, N. Topham and S. Furber, *Introducing SLAMBench, a performance and accuracy benchmarking methodology for SLAM*, IEEE Intl. Conf. on Robotics and Automation (ICRA 2015), Seattle, Washinton, May 2015.

[3] R. Potter, P. Keir, R. J. Bradford and A. Murray. *Kernel Composition in SYCL*. In ACM Proceedings of the International Workshop on OpenCL, 2015.

[4] L. Howes and M. Rovatsou (Eds.) *The Khronos SYCL Specification version 1.2*, 2015.

[5] J. Hoberock, editor. *Technical Specification for C++ Extensions for Parallelism*, ISO 2015

[6] R. Keryell. *triSYCL github*. `https://github.com/amd/triSYCL`, 2015.

[7] R. Reyes. *SyclParallelSTL github*. `https://github.com/KhronosGroup/SyclParallelSTL`, 2015.

[8] *Codeplay - ComputeCpp*. `https://www.codeplay.com/products/computecpp`, 2015.

[9] T. Drummond, E. Rosten, G. Reitmayr, G. Klein and Q. Pan. TooN: Tom's Object-oriented numerics library. `http://www.edwardrosten.com/cvd/toon/html-user`, 2015.

[10] A. Handa, T. Whelan, J. McDonald, and A. Davison. *A Benchmark for RGB-D Visual Odometry, 3D Reconstruction and SLAM*. In ICRA, 2014.

[11] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. *KinectFusion: Real-time dense surface mapping and tracking*. In ISMAR, 2011.

[12] G. Reitmayr and H. Seichter. *KFusion github*. `https://github.com/GerhardR/kfusion`, 2011.

[13] Wen-mei W. Hwu, editor. *GPU Computing Gems: Emerald Edition*. Chapter 26. Morgan Kaufmann, Amsterdam, 2011.

[14] K. Rupp, F. Rudolf and J. Weinbub. *ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs*, Proceeedings of the International Workshop on GPUs and Scientific Applications, 2010.